Alcyone: la boite a outils objets

J.M. Hullot

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports Techniques

N° 60

ALCYONE: LA BOÎTE À OUTILS OBJETS

Jean-Marie HULLOT

Novembre 1985

Alcyone
La Boîte à Outils Objets

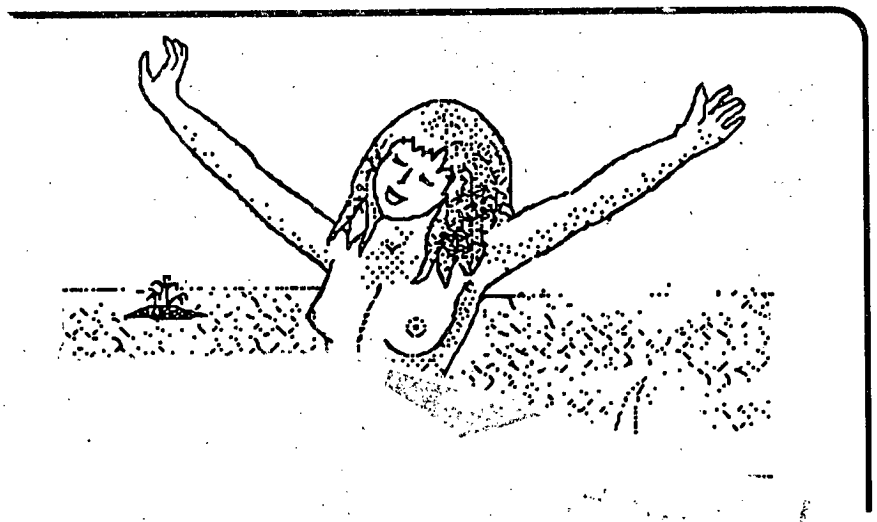
Jean-Marie Hullot

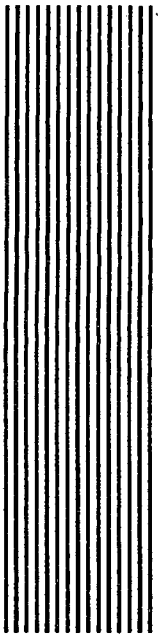
INRIA

Résumé: Nous décrivons dans ce papier un ensemble de primitives qui ajoutent la notion d'objets à Lisp.

Abstract: We describe in this paper a set of primitives that add objects to Lisp.

Alcyone





La composition de ce document a été réalisée sur Macintosh™ à l'aide des logiciels MacDraw™ et MacPaint™.

L'impression a été effectuée sur LaserWriter Apple™.

Alcyone

La Boite à Outils Objets

Jean-Marie Hullot *Août 1985*

Historique

Alcyone est le fruit d'une recherche sur les Langages Orientés Objets entreprise à l'INRIA depuis 1981.

La motivation première de cette recherche était de fournir au programmeur Lisp un mécanisme de structuration analogue aux records de Pascal et aux structures de C. Ceci fut réalisé dans les premières versions du langage Ceyx avec l'introduction de la notion de modèles.

Ceyx devint ce qu'il est convenu d'appeler un langage orienté objet, avec l'introduction de la hiérarchisation des espaces de modèles. Une première maquette introduisant les notions d'espace de noms et d'envoi de messages fut écrite en LeLisp.

Après une première étape de validation de la maquette, ces concepts furent introduits dans le noyau de Le Lisp avec le sytème de packages et la construction send. Les dernières versions de LeLisp possèdent aujoud'hui une version très efficace de la construction send.

Un grand nombre de fonctionnalités furent ensuite ajoutées à Ceyx pour en faire un langage orienté objet adapté aux besoins d'une communauté d'utilisateurs variés.

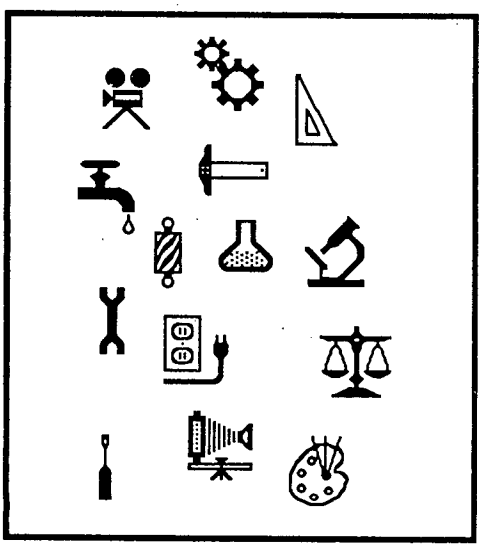
Avec l'accroissement du nombre des utilisateurs vint un grand nombre de propositions d'extension, principalement d'introduction de constructions existant dans d'autres langages orientés objets. Typiquement, chaque utilisateur réclamait un langage orienté objet taillé sur mesure pour son application.

La raison commanda alors de faire machine arrière et de ne plus continuer à étendre Ceyx, mais au contraire d'en extraire la quintessence.

C'est le but poursuivi avec Alcyone La Boite à Outils Objets. Avec Alcyone, vous avez toute la base de la programmation objets et tous les outils pour tailler sur mesure votre propre Langage Objet.

Alcyone

La Boîte à Outils Objets



Les bases de la programmation objets

Les outils pour tailler sur mesure votre propre langage objet tel Ceyx.

• Plan de Lecture

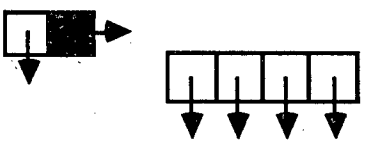
Installation

• Comment installer Alcyone sur votre disquette LeLisp.

• Comment tester le sytème en lançant les programmes de démonstrations.

Les Objets Lisp

• Rappels sur les structures de données élémentaires du langage Lisp.



Les Structures

• Définition de nouvelles structures de données.

• Définition de fonctions de manipulation de ces structures.

• Héritage.

Abréviations

• Pour faciliter l'écriture

Installation

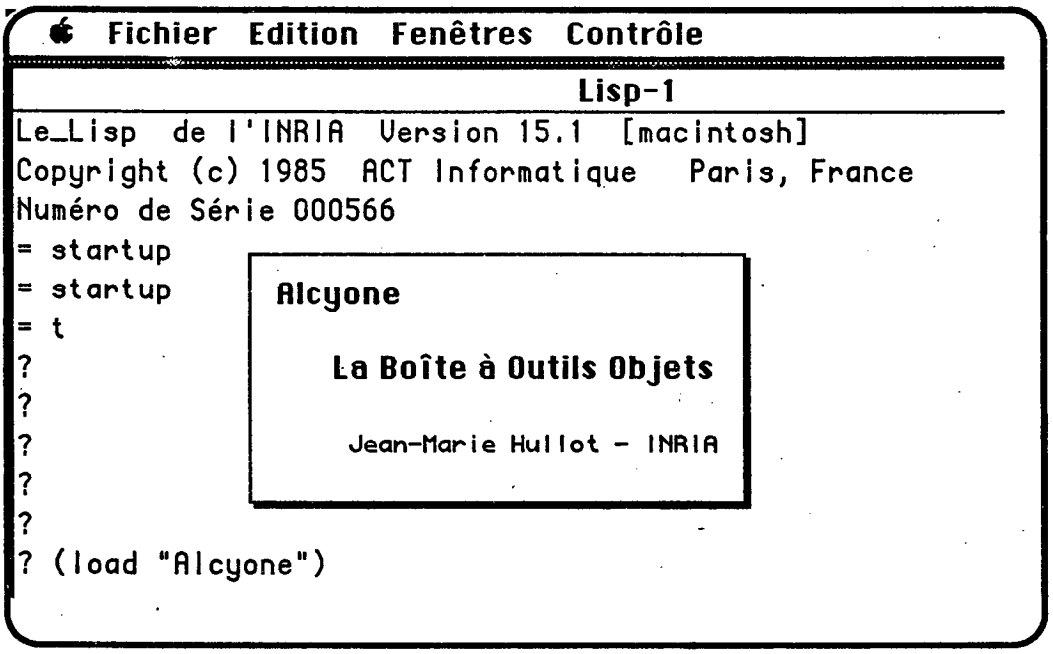
- Installation
- Démonstrations

• Installation

Pour disposer de Alcyone sous LeLisp, il suffit de charger le fichier Alcyone de la disquette Alcyone par la commande

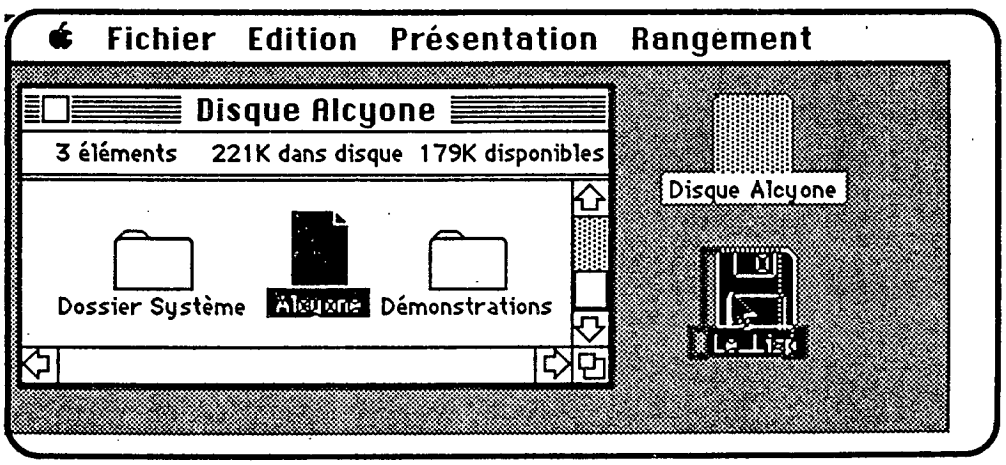
```
(load "Alcyone")
```

ceci à condition d'avoir préalablement déclaré la disquette Alcyone comme disquette par défaut (alinéa "Disque par Défaut" du menu Fichier).



Pour une installation permanente et si vous avez une place suffisante sur votre disquette LeLisp, transférez le fichier Alcyone sur la disquette LeLisp et rajoutez la ligne suivante dans le fichier startup de cette disquette:

```
(load "Alcyone")
```



• Démonstrations

Il existe un certain nombre de programmes de démonstrations sur la disquette Alcyone. Tous ces programmes sont rassemblés dans le dossier de nom Démonstrations. Pour les charger il suffit de charger le fichier 'Demos' en tapant au niveau Lisp:

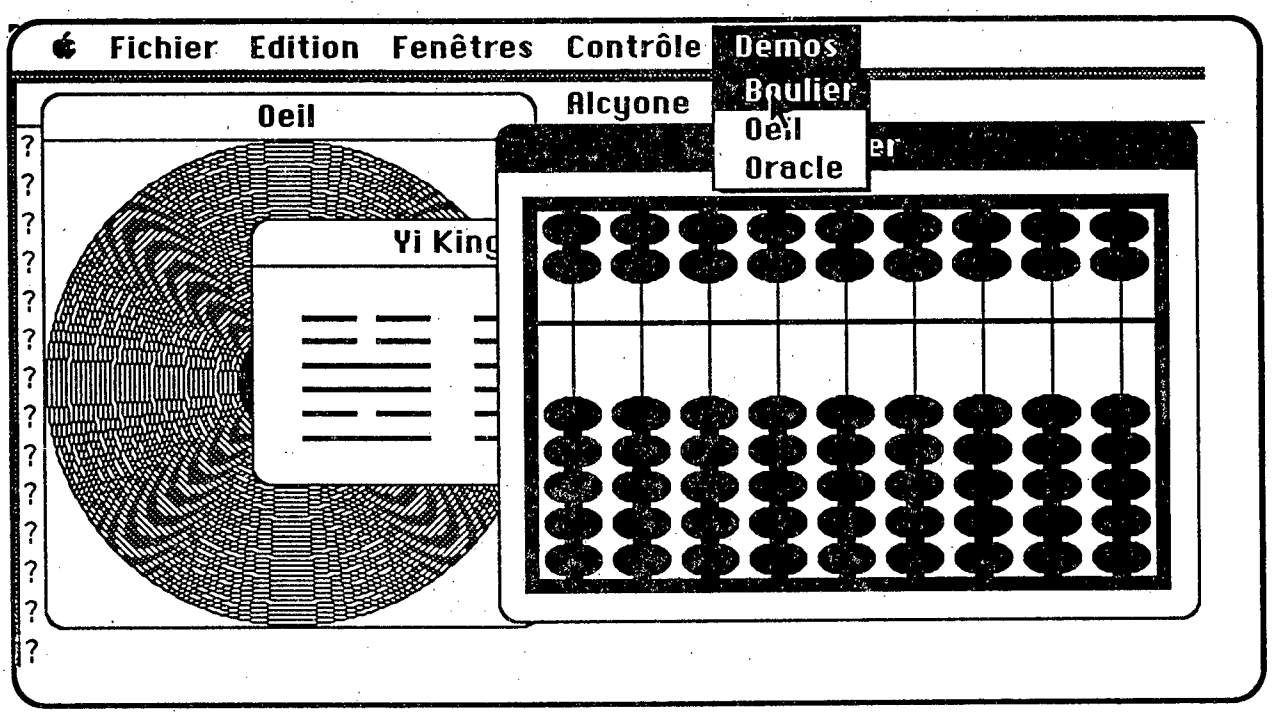
```
(load "Demos")
```

ceci à condition d'avoir déclaré la disquette Alcyone comme disquette par défaut (alinéa "Disque par Défaut" du menu Fichier).

Un nouveau menu de nom Démos est alors ajouté à la barre de menu, il n'y a plus qu'à choisir une démonstration.

Pour sortir d'une démonstration, il suffit de cliquer en dehors de la fenêtre qui lui est allouée.

Vous pouvez bien sûr ouvrir les fichiers de démonstrations à l'aide de l'éditeur de textes et regarder le code de ces démonstrations.

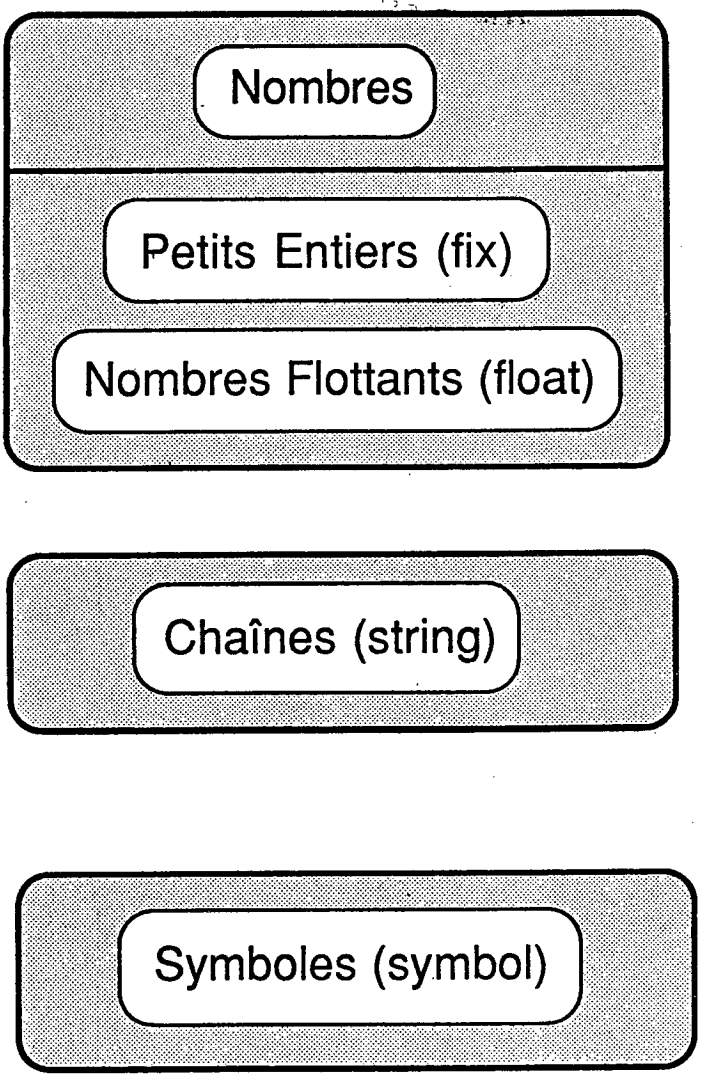


Exemple - Le Boulier - Pour faire vos calculs à la chinoise, cliquez sur les boules, pour remettre à zéro, secouez le boulier en cliquant sur son cadre.

Les Objets Lisp

- Les Objets atomiques
- Les Espaces de noms
- Les Cellules de listes
  - Créer une instance
  - Lire la valeur d'un champ
  - Modifier la valeur d'un champ
- Les Vecteurs
  - Créer une instance
  - Lire la valeur d'un champ
  - Modifier la valeur d'un champ
- Accéder au type d'un objet
- Les Objets Autotypés
  - Créer une instance
  - Evaluer une instance
  - Imprimer une instance

• Les Objets atomiques



• Le Lisp standard permet de manipuler deux types de nombres: les entiers signés sur 16 bits (fix) et les nombres approchés ou flottants (float).

Lecture	1, 1024, -58, 42
Evaluation	S'évalue à lui-même
Impression	Identique à la lecture

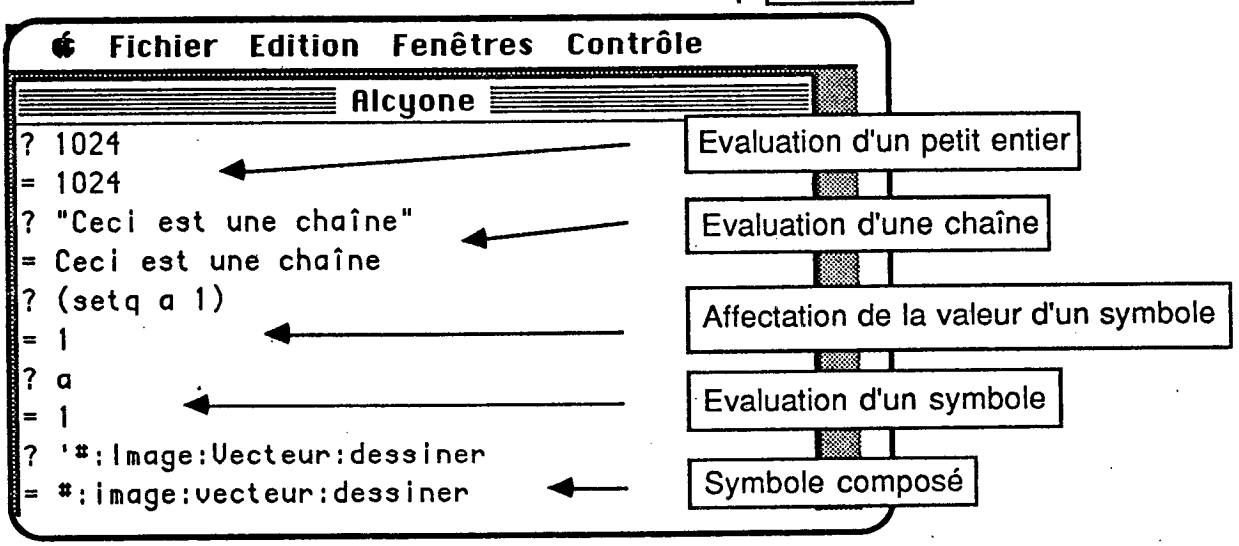
Lecture	1.414, 1.0545E-27
Evaluation	S'évalue à lui-même
Impression	Identique à la lecture

• Les chaînes de caractères sont des suites de caractères distincts du guillemet ". La lecture se fait entre guillemets, l'impression sans les guillemets.

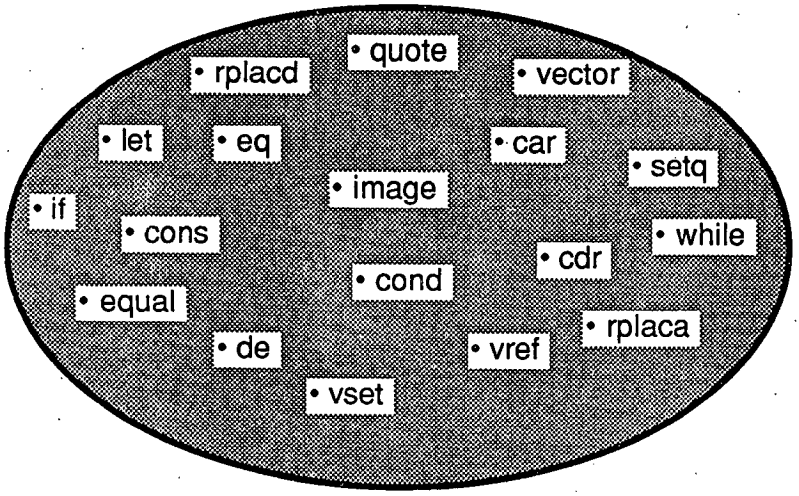
Lecture	"Ceci est une chaîne"
Evaluation	S'évalue à elle-même
Impression	Ceci est une chaîne

• Les symboles sont les identificateurs du langage. Ils servent à nommer des variables, des fonctions et des structures. Ils sont lus et écrits comme des suites de caractères non spéciaux (espace, passage à la ligne, guillemet, dièze, ...) ou d'une manière plus sophistiquée dans le cas de symboles composés (voir page suivante).

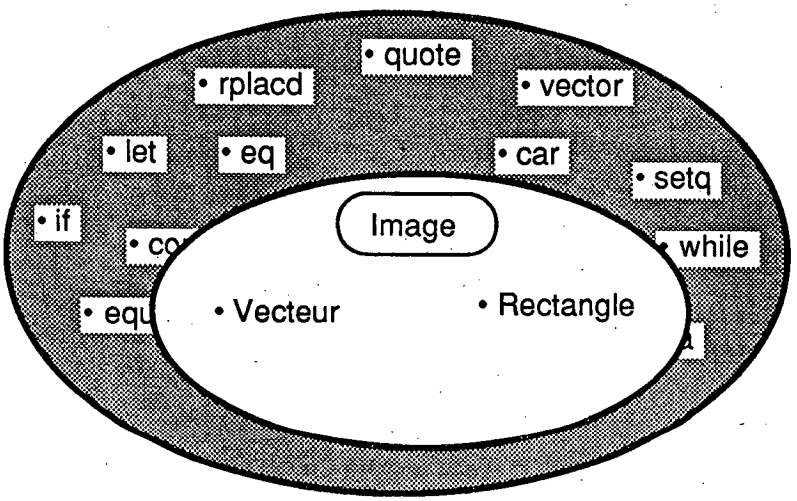
Lecture	de, setq, Image
Evaluation	Suivant l'environnement
Impression	de, setq, image



• Les Espaces de noms



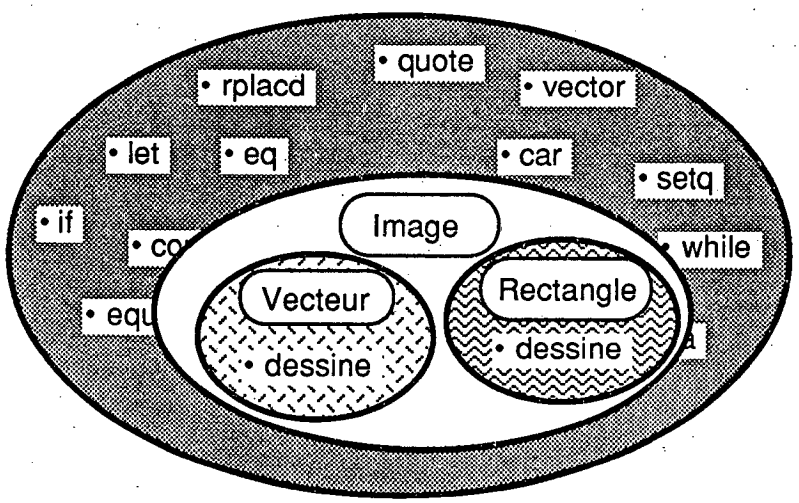
• Les symboles Lisp sont rassemblés dans une grande liste appelée l'oblist. Chaque fois qu'un symbole est lu il est placé dans l'oblist, s'il n'y était pas déjà. Ainsi il n'existe qu'un seul symbole d'un nom donné.



• En Le Lisp l'oblist n'est pas organisée de manière linéaire, mais de manière hiérarchique. En d'autres termes, l'oblist n'est pas une liste mais un arbre.
A chaque symbole est canoniquement associé un espace de noms.
Pour créer le symbole de nom Vecteur dans l'espace de noms du symbole Image, on écrit:

#:Image:Vecteur

c'est à dire qu'on spécifie tout à la fois l'espace de noms et le nom du symbole.



• Il est bien sûr possible de créer des symboles à n'importe quel niveau dans la hiérarchie. Ainsi pour créer les symboles de nom 'dessine' dans les espaces de noms de

#:Image:Vecteur, et
#:Image:Rectangle

il suffit d'écrire:

#:Image:Vecteur:dessine, et
#:Image:Rectangle:dessine.

- **Les Cellules de listes (cons)**
- **Créer une instance**

(cons 1 2)

- **Lire la valeur d'un champ**

(car objet)

(cdr objet)

- **Modifier la valeur d'un champ**

(rplaca objet 3)

(rplacd objet 4)

• La première structure composite de Lisp est la cellule de liste qui permet de regrouper deux objets en une paire.

Lecture	(1 . 2), ((a . b) . c)
Evaluation	appel de fonction
Impression	Identique à la lecture

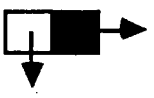
Le lecteur, comme l'imprimeur, autorise les abréviations suivantes:

(a . ()) -> (a)

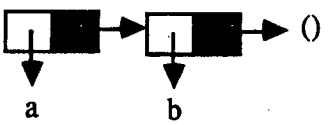
(a . (b . ...)) -> (a b . ...)

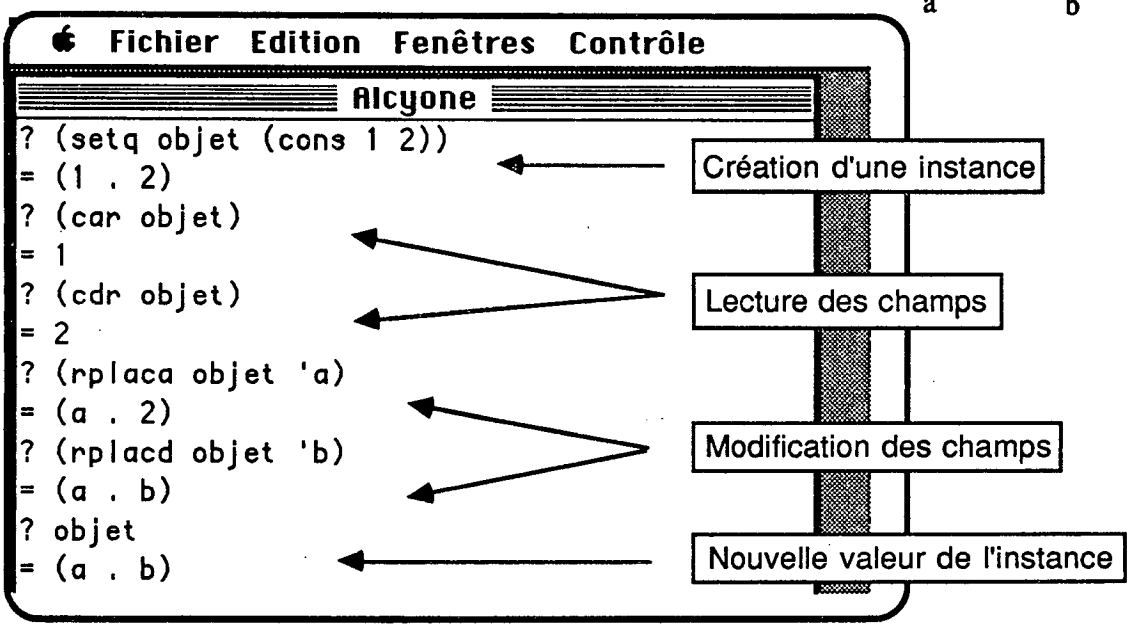
où () désigne la valeur du symbole spécial nil.

Les cellules de listes sont habituellement représentées graphiquement par



où les flèches représentent les pointeurs vers les parties car et cdr. Ainsi, la liste (a b) est-elle repésentée par:





- **Les Vecteurs (vector)**

- **Créer une instance**

(vector 4 5 6 7)

- **Lire la valeur d'un champ**

(vref objet 3)

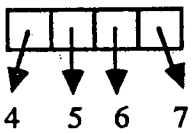
- **Modifier la valeur d'un champ**

(vset objet 2 10)

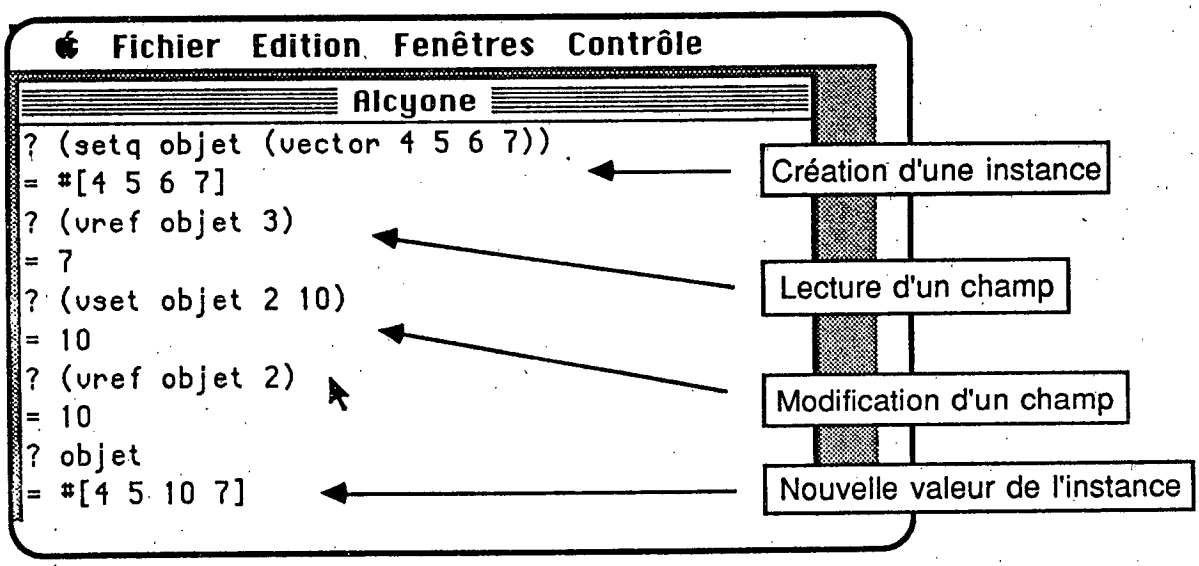
• Il est également possible de regrouper un nombre fini d'objets grâce à la structure de vecteur.

Lecture	#[4 5 6 7]
Evaluation	S'évalue à lui-même
Impression	Identique à la lecture.

Les vecteurs sont habituellement représentés graphiquement par



où les flèches représentent les pointeurs vers les différentes composantes du vecteur.



• Accéder au type d'un objet

(type-of objet)

• Six classes d'objets ont été définies (fix, float, string, symbol, cons, vector). Nous appelons type de l'objet le nom de la classe à laquelle il appartient.
La fonction type-of ramène en valeur le type de l'objet passé en argument. Cette fonction reconnaît à la fois les six types primitifs et les types utilisateurs définis ci-dessous.

• Les Objets Autotypés

• De manière à donner à l'utilisateur la possibilité de définir lui-même ses propres types, Le Lisp introduit une troisième structure composite: les cellules de liste étiquetées (tcons). Leurs fonctions d'accès sont identiques à celles du cons ordinaire (car, cdr, rplaca, rplacd).

• Créer une instance

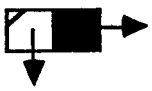
(tcons 'Point (vector 10 15))

Type de l'objet

Représentation de l'objet

Lecture	#(Point . #[10 15])
Evaluation	Par défaut: Identique au cons
Impression	Par défaut: Identique à la lecture

Les cellules de liste étiquetées sont habituellement représentées graphiquement par:



• Les objets formés au moyen d'une cellule de liste étiquetée sont appelés objets autotypés.
Leur partie car doit contenir un symbole qui est considéré comme le type de l'objet: la fonction type-of renvoie toujours ce symbole en valeur pour les objets autotypés.
Leur partie cdr contient un objet Lisp quelconque qui est appelé la représentation de l'objet.

• Evaluer une instance

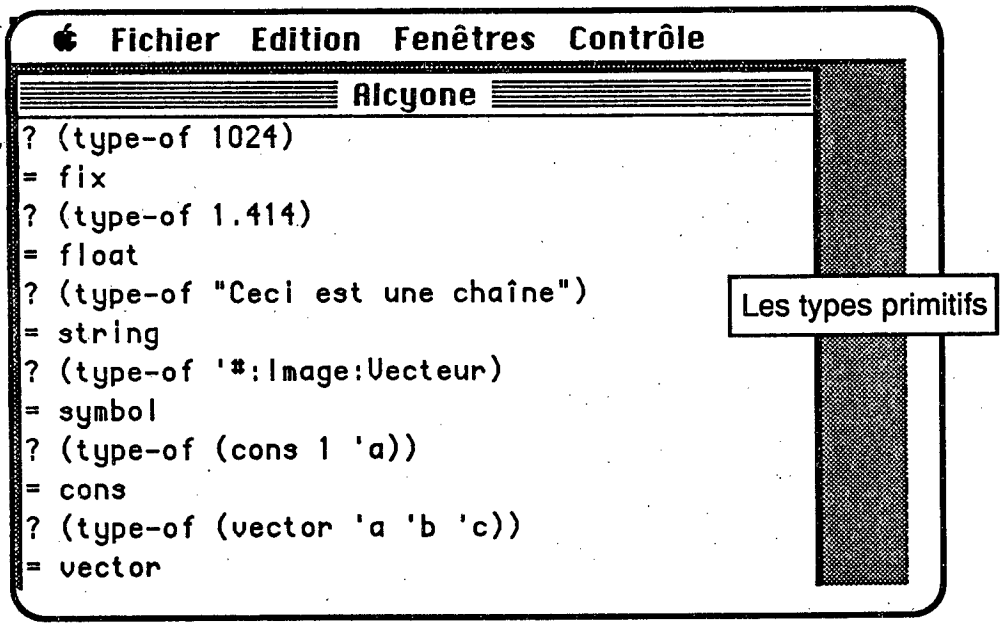
(de #:Point:eval (point) . . .)

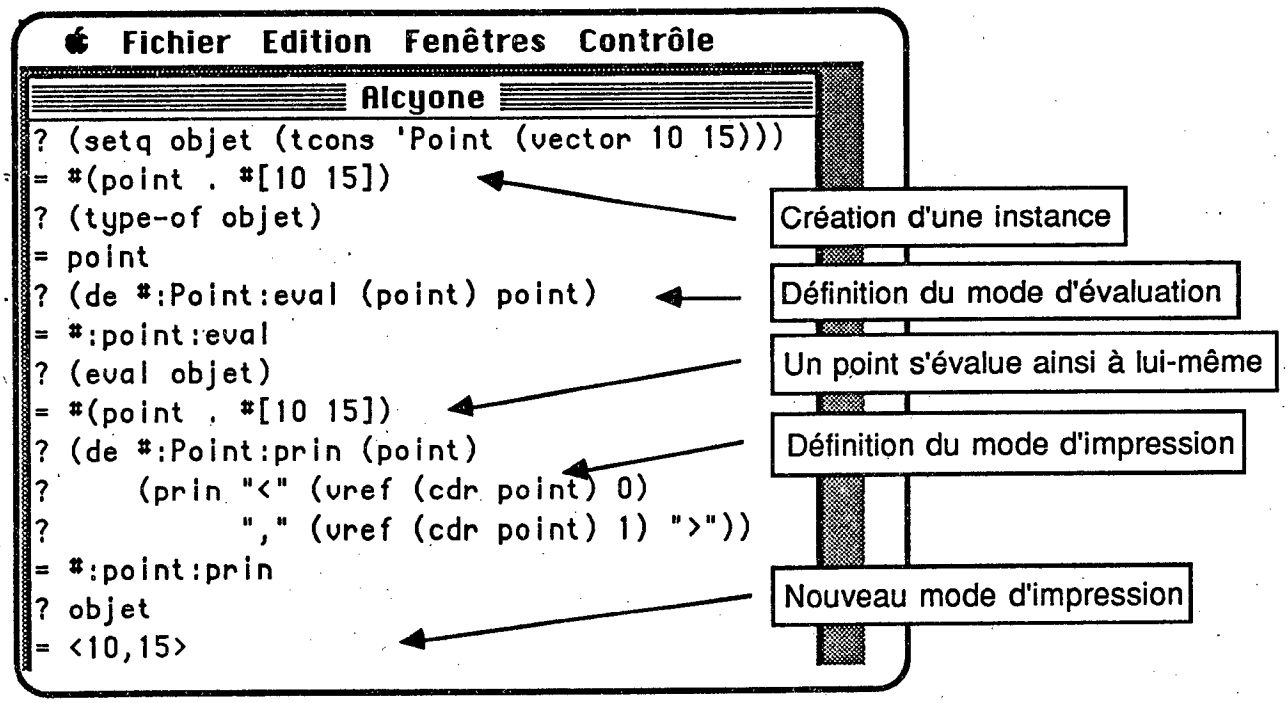
Les objets autotypés sont reconnus par l'évaluateur et l'imprimeur Lisp. Pour changer le mode d'évaluation ou d'impression pour un type donné, il suffit de définir les fonctions eval ou prin dans l'espace de noms canoniquement associé au nom du type.

• Imprimer une instance

(de #:Point:prin (point) . . .)

• Les objets autotypés seront le plus souvent utilisés par l'intermédiaire des structures qui seront présentées ultérieurement.


Fichier Edition Fenêtres Contrôle
Alcyone
? (type-of 1024)
= fix
? (type-of 1.414)
= float
? (type-of "Ceci est une chaîne")
= string
? (type-of '#:Image:Vecteur)
= symbol
? (type-of (cons 1 'a))
= cons
? (type-of (vector 'a 'b 'c))
= vector
Les types primitifs


Fichier Edition Fenêtres Contrôle
Alcyone
? (setq objet (tcons 'Point (vector 10 15)))
= #(point . #[10 15])
? (type-of objet)
= point
? (de #:Point:eval (point) point)
= #:point:eval
? (eval objet)
= #(point . #[10 15])
? (de #:Point:prin (point)
? (prin "<" (vref (cdr point) 0)
? "," (vref (cdr point) 1) ">"))
= #:point:prin
? objet
= <10,15>
Création d'une instance
Définition du mode d'évaluation
Un point s'évalue ainsi à lui-même
Définition du mode d'impression
Nouveau mode d'impression

Les Structures

- Définir une Structure
- Créer une instance
- Lire la valeur d'un champ
- Modifier la valeur d'un champ
- Définir une méthode
- Envoyer un message
- Etendre une structure
- Hériter une méthode
- Masquer une méthode

• Définir une Structure

```
(defstruct Cadre
           (gauche 0)
           (haut 0)
           (droite 10)
           (bas 10))
```

• Associe au symbole Cadre, la définition de structure à quatre champs de noms gauche, haut, droite, bas.

Les valeurs par défaut de ces champs, c'est-à-dire les valeurs posées lors d'une création par (new 'Cadre), sont respectivement 0, 0, 10 et 10.

Si une valeur par défaut est omise, elle est prise égale à nil.

Exemple:

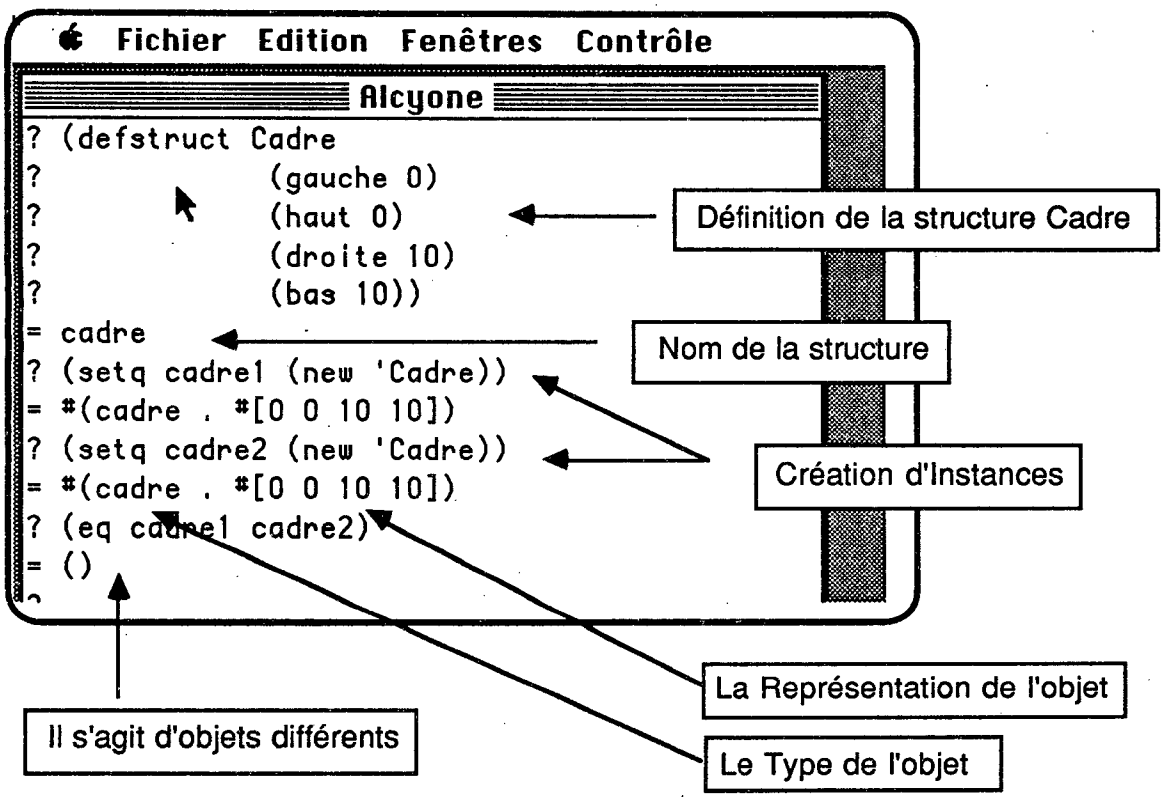
```
(defstruct Cadre
    gauche haut
    droite bas)
```

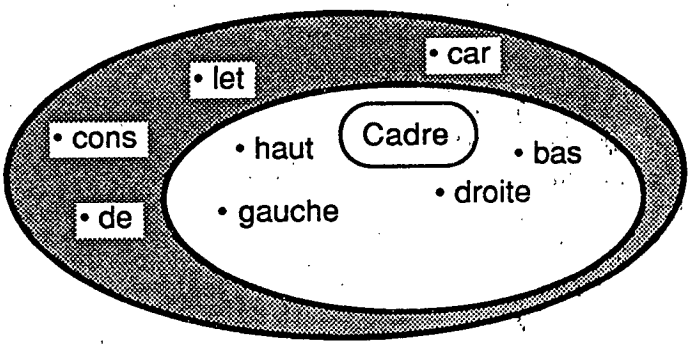
• Créer une instance

```
(new 'Cadre)
```

• Ramène en valeur une instance de la structure de nom Cadre, c'est-à-dire un objet autotypé de type Cadre, ayant pour représentation un vecteur de longueur 4.

Pour accéder aux valeurs des différents champs de cet objet, on utilisera les fonctions d'accès aux champs qui sont automatiquement engendrées (voir page suivante).





• Lors de la définition d'une structure par defstruct, une fonction d'accès est définie pour chacun des champs. Cette fonction porte le nom du champ dans l'espace de noms de la structure. Par exemple, pour le champ bas:

#:Cadre:bas

• Lire la valeur d'un champ

```
(#:Cadre:gauche cadre1)

(#:Cadre:bas cadre2)
```

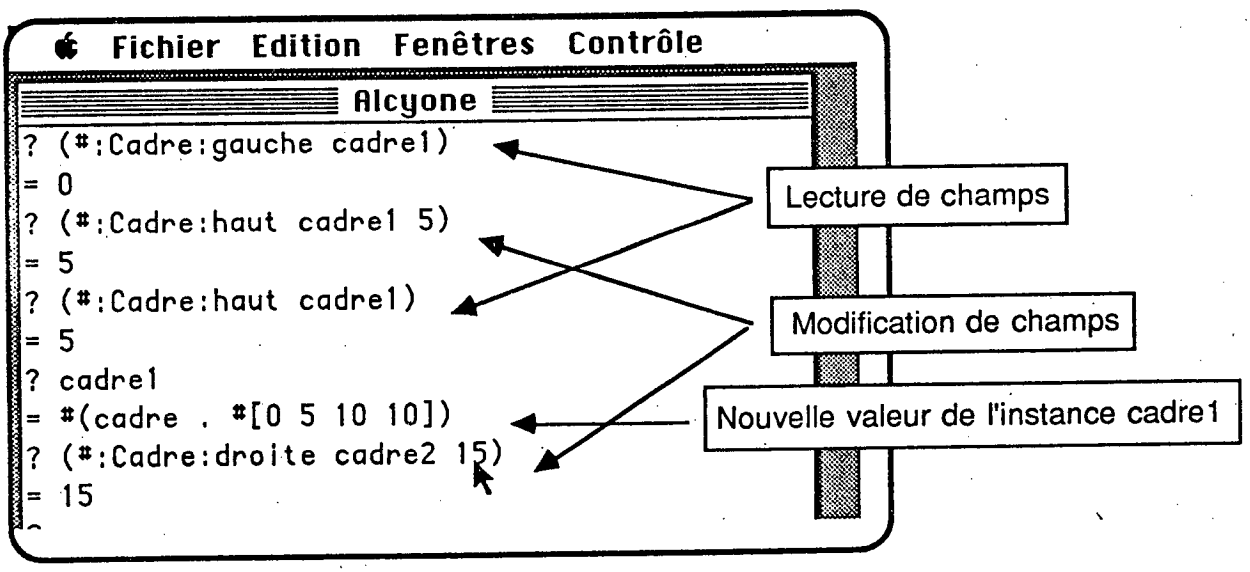
• Utilisées avec un argument seulement, qui doit être instance d'une structure du type voulu, ces fonctions renvoient en valeur la valeur du champ correspondant pour cette instance.

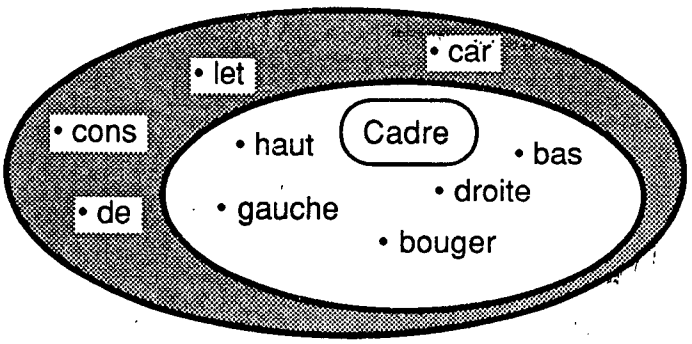
• Modifier la valeur d'un champ

```
(#:Cadre:haut cadre1 5)

(#:Cadre:droite cadre2 15)
```

• Utilisées avec deux arguments, elles remplacent la valeur du champ correspondant du premier argument par la valeur du second.





• Les Méthodes d'une structure sont ses fonctions de manipulation spécifiques.

Toutes les méthodes d'une structure sont définies dans l'espace de noms associé à la structure.

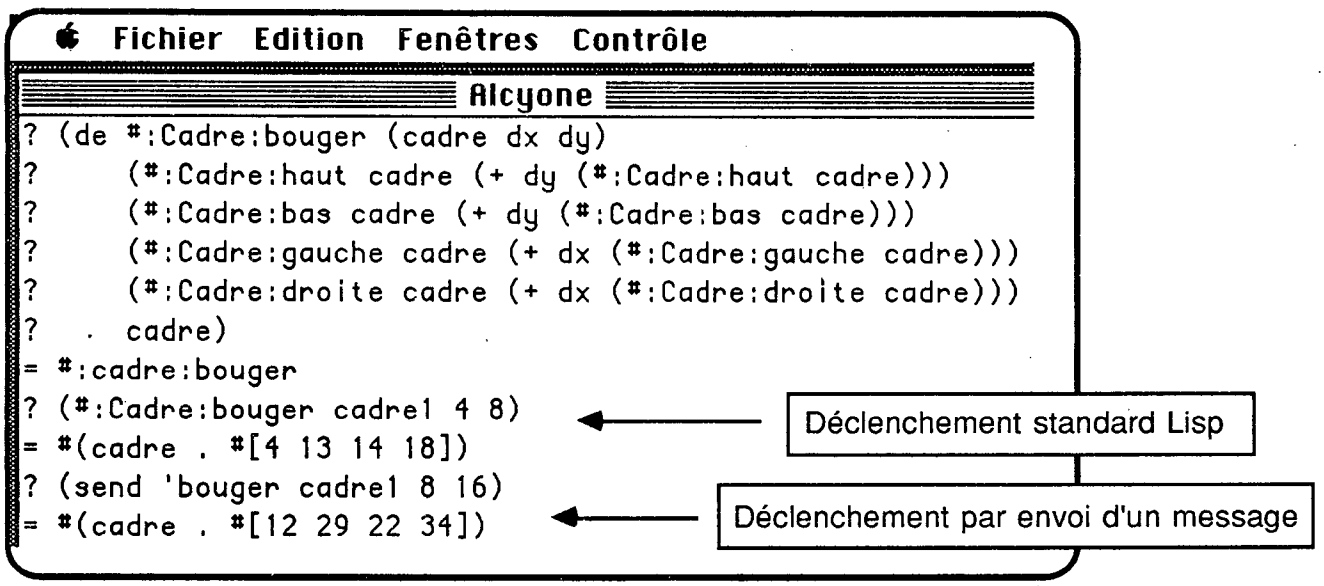
Les fonctions d'accès aux champs sont des cas particuliers de méthodes.

• Définir une Méthode

```
(de #:Cadre:bouger (cadre dx dy)
    (#:Cadre:haut cadre
        (+ dy (#:Cadre:haut cadre)))
    (#:Cadre:bas cadre
        (+ dy (#:Cadre:bas cadre)))
    (#:Cadre:gauche cadre
        (+ dx (#:Cadre:gauche cadre)))
    (#:Cadre:droite cadre
        (+ dx (#:Cadre:droite cadre)))
  cadre)
```

• Définir une méthode pour une structure, consiste à définir une fonction dans l'espace de noms associé à la structure, fonction dont le premier argument est une instance de la structure. C'est à cet objet que seront envoyés les messages par la construction send.

Les méthodes peuvent être invoquées soit comme des fonctions Lisp standards, soit à l'aide de la construction send présentée ci-contre.



• Envoyer un message

(send 'bouger cadre1 8 16)

• Cette construction est la construction fondamentale du sytème. Elle donne une alternative au mécanisme d'appel de fonction en Lisp. Au lieu d'appliquer une fonction à une liste d'arguments:

(#:Cadre:bouger cadre1 8 16)

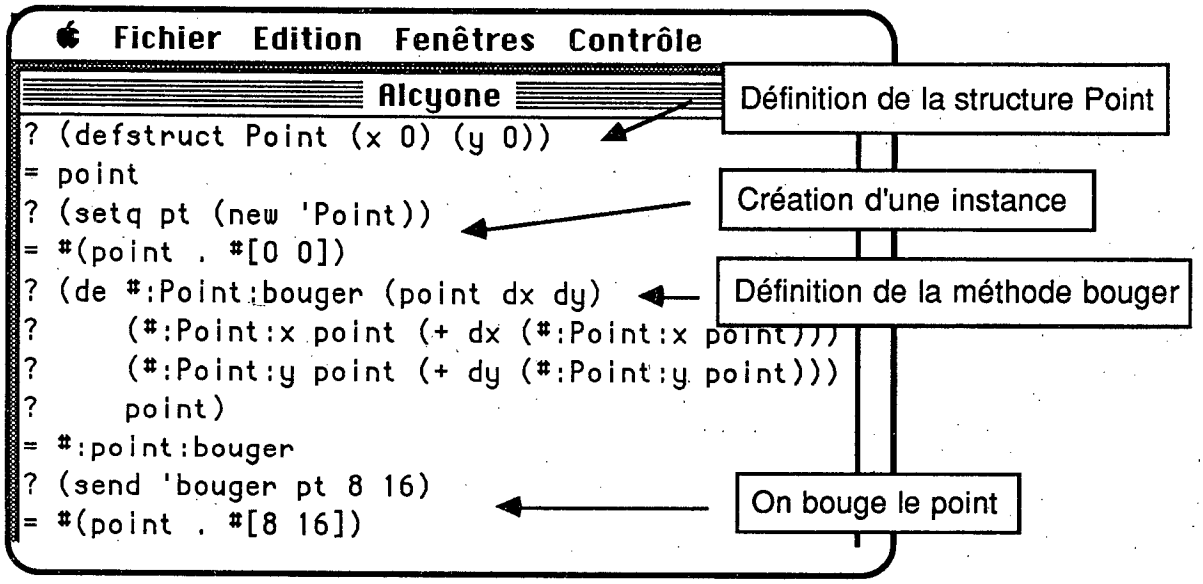
on envoie à une instance de structure un message avec une liste d'arguments:

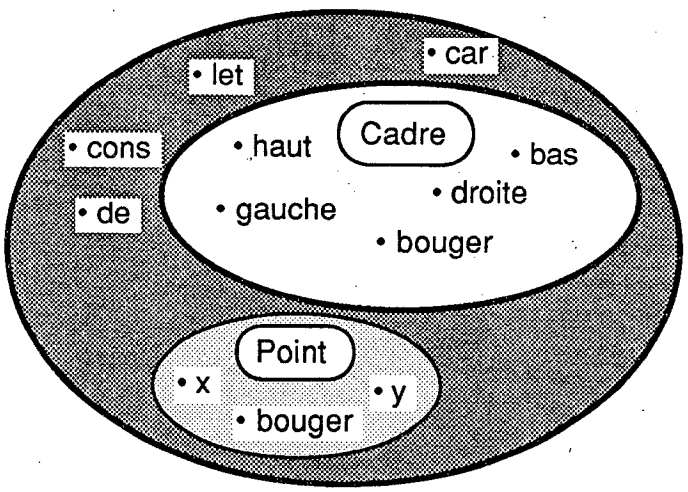
(send 'bouger cadre1 8 16).

La différence essentielle est que le système décide de lui-même, compte tenu du type de l'instance et du nom du message, la fonction à appliquer: c'est la fontion portant le nom du message dans l'espace de noms associé au type. De cette manière, on bougera tout objet par le même appel:

(send 'bouger objet 8 16)

dans la mesure où la méthode bouger a été définie pour ce type d'objet.





• Les espaces de noms après définition de la structure Point et de sa méthode bouger.

• Etendre une Structure

```
(defstruct #:Cadre:Fenêtre
           (titre "sans-titre")
           contenu)
```

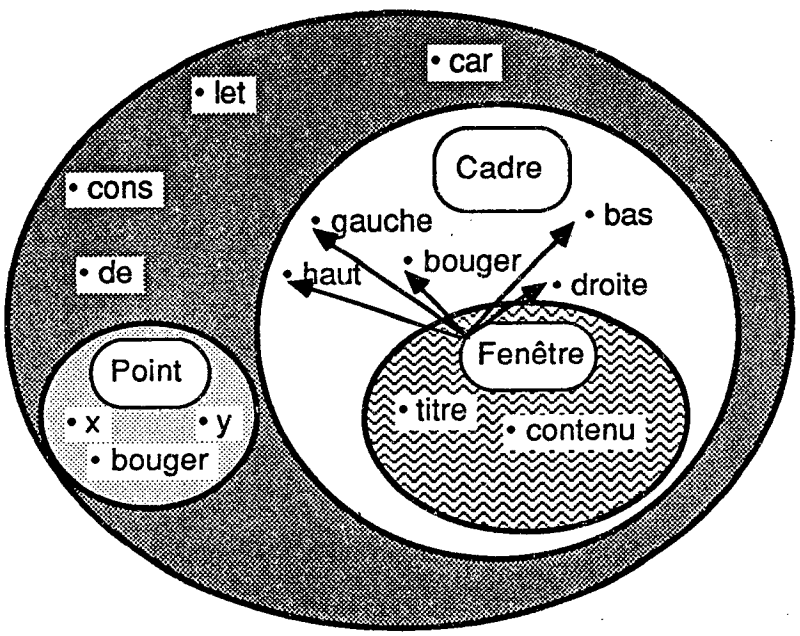
• Etendre une structure, c'est en définir une nouvelle qui possède tous les champs de la première plus des champs propres.

On réalisera cette opération à l'aide de la construction defstruct en définissant la nouvelle structure dans l'espace de noms de l'ancienne et en indiquant seulement les nouveaux champs.

Ainsi #:Cadre:Fenêtre est-elle une structure possédant les champs gauche, haut, bas, droite (qui sont hérités), et titre et contenu.

• Hériter une méthode

```
(send 'bouger fenêtre 10 10)
```



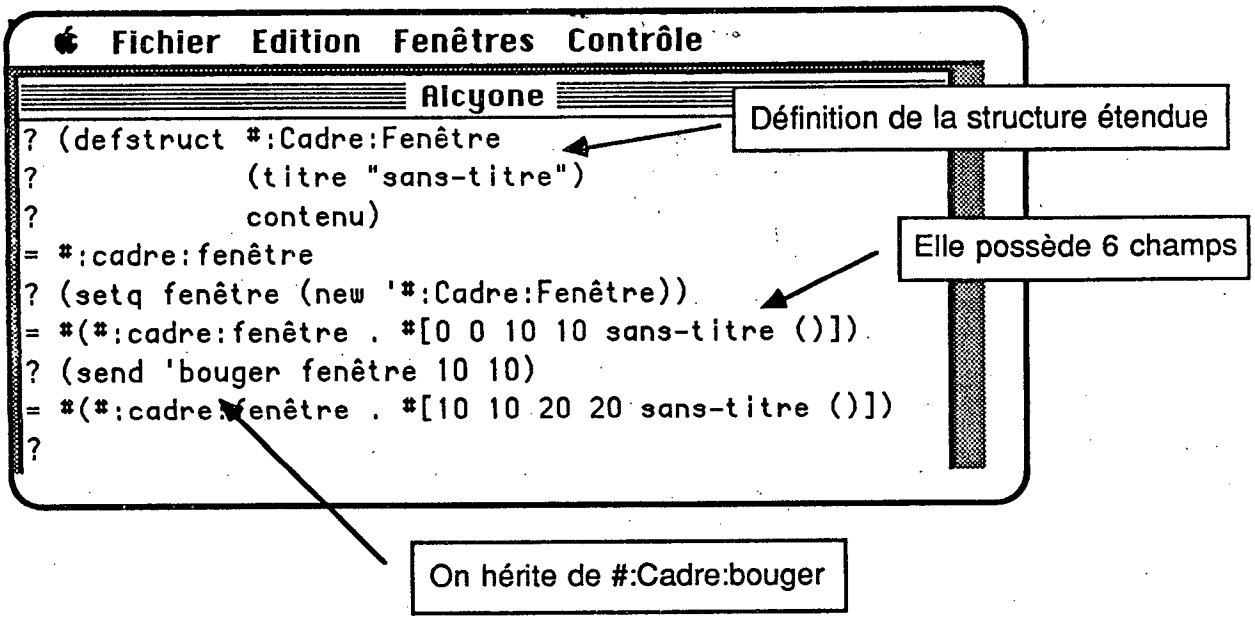
• L'un des intérêts majeurs de la construction send réside dans le fait qu'elle réalise la recherche de la fonction à appliquer en remontant dans l'arbre des espaces de noms associés aux structures.

Ainsi, si l'on envoie le message bouger à un objet de type fenêtre:

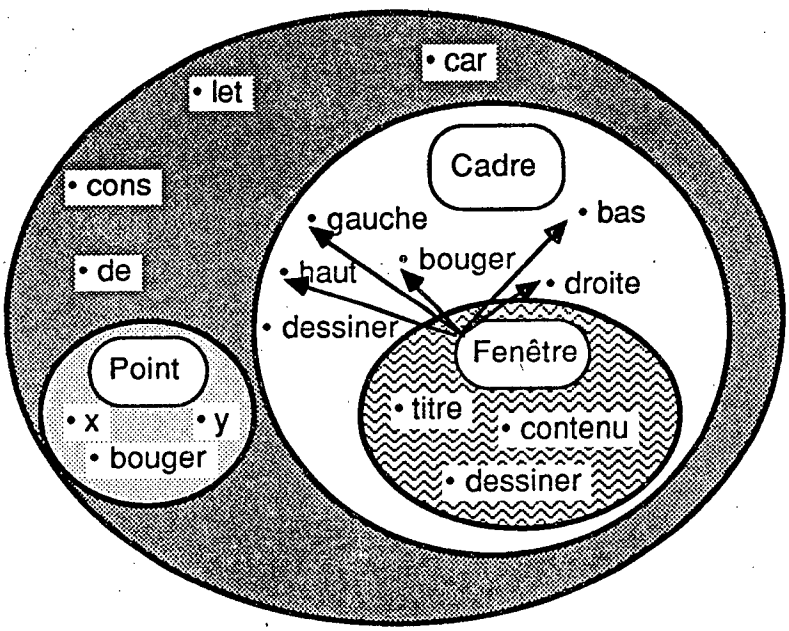
(send 'bouger fenêtre 10 10)

la recherche dans l'espace de noms de #:Cadre:Fenêtre échoue, mais elle est poursuivie dans l'espace de noms de Cadre où l'on découvre la fonction à appliquer:

#:Cadre:bouger



• Masquer une méthode



• Supposons avoir défini une méthode dessiner pour les cadres qui se contente de tracer le contour du cadre:

(de #:Cadre:dessiner (cadre)
. . .)

On ne dessine pas un cadre comme on dessine une fenêtre: il faut en plus écrire le titre et visualiser le contenu. Pour éviter d'hériter de la méthode dessiner des Cadres nous définissons la méthode dessiner pour les fenêtres:

(de #:Cadre:Fenêtre:dessiner
(fenêtre)
. . .)

L'envoi du message dessiner à une instance de Fenêtre amènera alors à l'application de cette fonction.

Abréviations

- Définir une fonction de création
- Définir une méthode

• Définir une fonction de création

```
(defmake Point CreerPoint (x y))
```

• Il est souvent utile de créer des instances de structure dont certains champs prennent des valeurs données. C'est le but de la construction defmake.

La forme ci-contre est une abréviation pour:

```
(de CreerPoint (x y)
   (let ((point (new 'Point)))
      (#:Point:x point x)
      (#:Point:y point y)
      point))
```

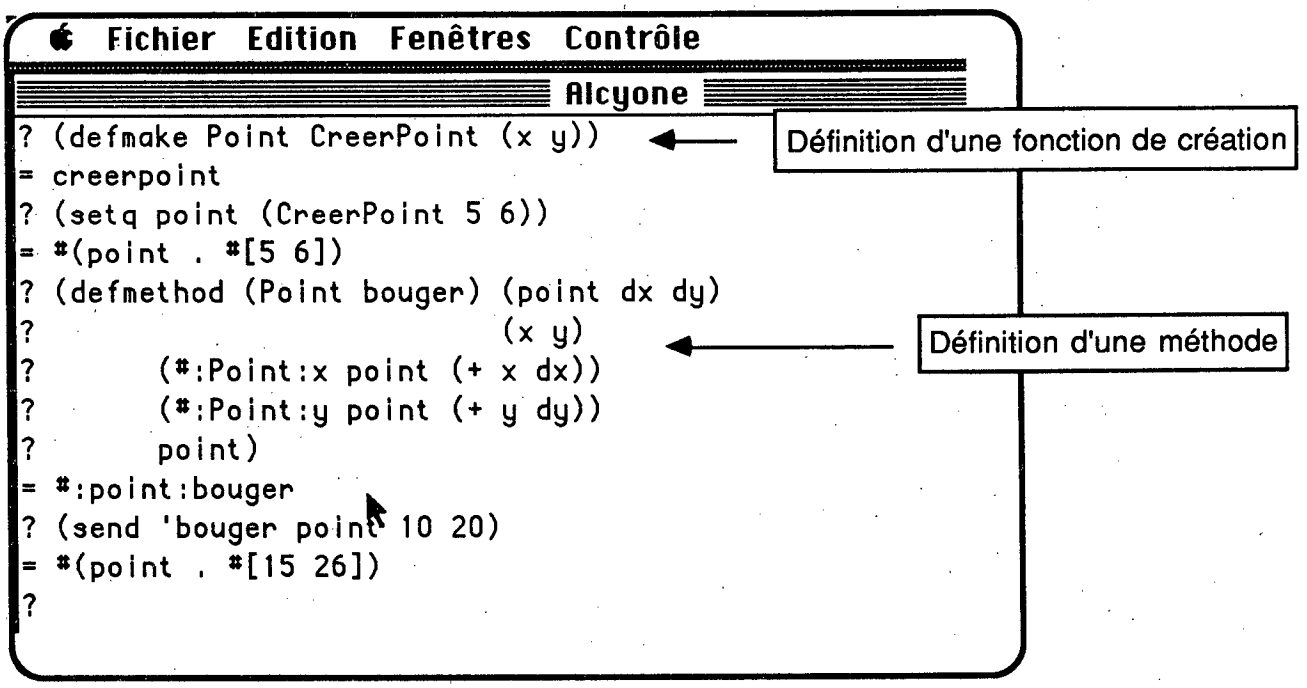
• Définir une méthode

```
(defmethod (Point bouger)
                (point dx dy ) (x y)
   (#:Point:x point (+ x dx))
   (#:Point:y point (+ y dy))
   point)
```

• La construction ci-contre est une abréviation pour:

```
(de #:Point:bouger (point dx dy)
   (let ((x (#:Point:x point))
         (y (#:Point:y point)))
      (#:Point:x point (+ x dx))
      (#:Point:y point (+ y dy))
      point))
```

Cette construction réalise donc une déstructuration d'un sous-ensemble donné des champs d'une instance de structure.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique